Efficiently Parallelizable Strassen-Based Multiplication of a Matrix by its Transpose

Viviana Arrigoni*
arrigoni@di.uniroma1.it
Department of Computer Science,
Sapienza University of Rome
Rome, Italy

Filippo Maggioli*
maggioli@di.uniroma1.it
Department of Computer Science,
Sapienza University of Rome
Rome, Italy

Annalisa Massini
massini@di.uniroma1.it
Department of Computer Science,
Sapienza University of Rome
Rome, Italy

Emanuele Rodolà
rodola@di.uniroma1.it
Department of Computer Science,
Sapienza University of Rome
Rome, Italy

ABSTRACT

The multiplication of a matrix by its transpose, $\mathbf{A}^T\mathbf{A}$, appears as an intermediate operation in the solution of a wide set of problems. In this paper, we propose a new cache-oblivious algorithm (ATA) for computing this product, based upon the classical Strassen algorithm as a sub-routine. In particular, we decrease the computational cost to $2/3$ the time required by Strassen's algorithm, amounting to $\frac{14}{3}n^{\log_2 7}$ floating point operations. ATA works for generic rectangular matrices, and exploits the peculiar symmetry of the resulting product matrix for saving memory. In addition, we provide an extensive implementation study of ATA in a shared memory system, and extend its applicability to a distributed environment. To support our findings, we compare our algorithm with state-of-the-art solutions specialized in the computation of $\mathbf{A}^T\mathbf{A}$. Our experiments highlight good scalability with respect to both the matrix size and the number of involved processes, as well as favorable performance for both the parallel paradigms and the sequential implementation, when compared with other methods in the literature.

KEYWORDS

Matrix Multiplication; Cache-oblivious algorithm; Fast Linear Algebra; MPI; OpenMP; C/C++.

1 INTRODUCTION

Matrix multiplication is a fundamental operation in Linear Algebra and HPC, as it appears as an intermediate step in a wide set of problems. Many researchers have devoted their efforts to the algorithmic aspects of matrix multiplication, with the aim of improving the computational cost of existing algorithms and to devise and implement new solutions for parallel architectures. Designing a distributed algorithm for matrix multiplication is a challenging task, due to the inherent dependence of the data scattered in the system's distributed memory, and due to the overhead due to the communication cost of assembling the resulting product matrix.

The product of a matrix by its transpose, $\mathbf{A}^T\mathbf{A}$ (as well as $\mathbf{A}\mathbf{A}^T$), is a particular matrix multiplication involved in several applications. For example, computing $\mathbf{A}\mathbf{A}^T$ is a straightforward, yet effective, method to check for orthogonality or to project vectors onto the

space spanned by the columns of $\mathbf{A}$. This product, in fact, is repeatedly computed in the Gram-Schmidt algorithm for vector basis orthogonalization, where $\mathbf{A}$ is the progressively built projection matrix. One way to solve the least squares problem of under and over determined linear systems $\mathbf{A}x = b$, is to solve the associated system of normal equations, obtained by left-hand multiplying the original system by $\mathbf{A}^T$, thus obtaining a square linear system $\mathbf{A}^T\mathbf{A}x = \mathbf{A}^Tb$. Also, the Singular Value Decomposition (SVD) of a matrix $\mathbf{A}$ can be computed by studying the eigenproblem for $\mathbf{A}^T\mathbf{A}$ and $\mathbf{A}\mathbf{A}^T$. Furthermore, the product of a matrix by its transpose commonly arises in discrete exterior calculus and discrete differential geometry. One example is given by the discrete heat kernel $\mathbf{K}(t) = \Phi\mathbf{E}(t)\Phi^T$, with $\mathbf{E}(t) = \exp(-\Lambda t)$ being a diagonal matrix, so that $\mathbf{K}(t) = (\Phi\mathbf{E}(t)^{1/2})(\Phi\mathbf{E}(t)^{1/2})^T$ can be efficiently computed [38]. Many other applications of the product $\mathbf{A}^T\mathbf{A}$ are described in [32], together with its properties such as positive semi-definiteness.

In this work, we consider the multiplication between $\mathbf{A}^T$ and $\mathbf{A}$, where $\mathbf{A}$ may have any size and shape. We rely on a recursive approach that, as described in [28], is virtually tuning free and avoids the significant tuning efforts that are required by iterative blocked algorithms to achieve near-optimal performance. Our contribution is threefold.

- First, we introduce ATA (Section 3), a cache-oblivious algorithm for computing $\mathbf{A}^T\mathbf{A}$ that requires $2/3n^{(\log_2 7)} + 1/3n^2$ multiplications. We exploit the self-similarity of the $\mathbf{A}^T\mathbf{A}$ product with its sub-problems and the Strassen's algorithm, that is recursively applied to possibly rectangular matrices, without introducing additional computational and space cost, deriving from dynamic peeling and padding, as in [22, 34]. In contrast to [13], our algorithm works on any algebraic field. We prove that ATA exhibits high efficiency for both memory and time, and show that it is efficiently implementable, as it does not hide large constant factors. We also describe our implementation of Strassen's algorithm, and compare its performance with that of the Intel MKL BLAS gemm routine for matrix multiplication.

- Second, we describe ATA-S, our multi-threaded implementation of ATA for a shared memory system, relying on OpenMP (Section 4.2). A well-engineered scheduler that assigns different tasks to each thread in such a way that computations can be carried out

---

*Both authors contributed equally to this research.

in perfect parallelism by preventing memory collisions. Performance evaluation shows that our implementation outperforms the multi-threaded Intel MKL BLAS routines (e.g. syrk for symmetric rank-K update) on large matrices, even on Intel processors.

- Finally, we extend our approach to distributed systems, leveraging the standard message-passing paradigm MPI. Our distributed algorithm AtA-D allows the distribution of the computational effort among a larger number of processes (Section 4.3). This is particularly convenient on very large matrices.

To validate the effectiveness of our algorithms, we study their performance by running a set of tests on dense matrices of variable size (Section 5). We analyse different metrics for evaluating the scalability of our parallel implementation, and compare our results with benchmark solutions for distributed systems. We run tests on a cluster of multi-core nodes endowed with 2×8 core Intel Xeon E5-2630v3 processors, 2.4 Ghz, 4 GB RAM/core.

2 RELATED WORK

Nowadays, matrix multiplication is still a hot topic in HPC and numerical algorithmics. In 1969, Strassen [33] was the first to reduce the computational complexity of the standard matrix multiplication from $O(n^3)$ to $O(n^{\log_2 7})$. More recently, Coppersmith and Winograd [9] devised an algorithm for matrix multiplication running in $\sim O(n^{2.38})$ time. In the last decade, many have devoted their efforts to improve this limit ([18, 31, 36]). These works make use of algebraic tensors that, despite the elegance of the resulting method, are still hardly used in practice as they come at the cost of very large hidden constants and frequent memory access.

Several authors have designed hybrid algorithms, deploying Strassen's multiplication in conjunction with conventional matrix multiplication, to overcome the overhead of Strassen's algorithm on small matrices, see, e.g., [4–6, 20, 22]. Huss-Lederman *et al.* [22] propose two techniques, known as dynamic peeling and static padding, in order to apply Strassen's algorithm to odd-sized matrices. Thottethodi *et al.* [34] propose two strategies to optimize memory efficiency in Strassen by minimizing padding and peeling operations. Many researchers have proposed a parallel implementation of Strassen's algorithm. In [27], Luo and Drake explored Strassen-based parallel algorithms that use the communication patterns known for classical matrix multiplication. They considered using a classical 2D parallel algorithm and using Strassen locally and at the highest level. This approach is improved in [19] by using a more efficient parallel matrix-multiplication algorithm running on a more communication-efficient machine. In [10], Strassen's algorithm is extended to deal with rectangular and arbitrary-size matrices. Their approach leverages on a suitable combination of Strassen's with ATLAS and GotoBLAS. Other parallel approaches [12, 21, 30] have used more complex parallel schemes and communication patterns, and consider at most two steps of Strassen. In [1], a parallel algorithm based on Strassen's fast matrix multiplication, Communication - Avoiding Parallel Strassen (CAPS), is described. The authors show that its complexity matches the communication lower bounds described in [2]. This work is extended in [11] to handle rectangular matrices (CARMA). More recently, Kwasniewski *et al.* [26] proposed a near optimal algorithm for matrix multiplication that models the matrix multiplication dependencies by the red-blue

pebble game [23] to derive an I/O optimal schedule, improving the performance of previous works.

Both Strassen's algorithm and AtA fall into the class of recursive blocked algorithms. The work in [15, 25] proves the effectiveness of this kind of algorithms for dense Linear Algebra. The work in [14] introduces FRPA, an interface for implementing recursive problems in parallel that gets as an input the recursive problem, and handles parallelization and auto-tuning automatically. Similarly to our approach, Charara *et al.* [8] propose block recursive matrix multiplication and linear solver algorithms. They show how recursion enhances data reuse and concurrency in GPUs. Differently from the work presented in this paper, they specialize on triangular matrices. In [7], the authors also adapt this blocking strategy to handle batched operations on small matrix sizes (up to 256) to stress the register usage and maintain data locality. In [28], Elmar and Bientinesi introduce ReLAPACK, a collection of recursive algorithms for dense Linear Algebra. While this work corroborates the recursive approach that we implement in our algorithms, it does not provide a routine specialized in the $\mathbf{A}^T\mathbf{A}$ product for general matrices. Instead, they propose a routine for the same multiplication only on triangular matrices. We highlight that the solutions proposed for the multiplication of a matrix by its transpose on triangular matrices (TRSYRK) is useful for many applications but cannot be applied on general matrices.

Although much research has been devoted to optimizing the implementation of parallel matrix multiplication, very few solutions have been proposed for the $\mathbf{A}^T\mathbf{A}$ multiplication. In [13], Dumas *et al.* propose an algorithm for the product $\mathbf{A}\mathbf{A}^T$ whose computational complexity is improved by a constant factor with respect to previously known reductions. This approach is applicable only to matrices lying in fields where skew-orthogonal matrices exist (e.g., $\mathbb{C}$ and finite fields of prime characteristics), which is not the case for $\mathbb{R}$ and $\mathbb{Q}$, that instead are important in many applications, such as the study of embedded systems, computational geometry and system simulations.

Except for some sporadic attempts to implement a method for distributing in a balanced way the workload for matrix multiplication among processes with the MapReduce programming model [24, 29], the approach that we implement here for the distributed parallel model has barely been investigated.

3 ATA

In this section, we describe our sequential recursive algorithm for the matrix multiplication $\mathbf{A}^T\mathbf{A}$, dubbed AtA, and we provide implementation details. We remark that our solution also works for the product $\mathbf{A}\mathbf{A}^T$. Yet, when row-major order is the default layout for array storage, the $\mathbf{A}^T\mathbf{A}$ multiplication is in practice harder to perform, as memory access is inherently column-wise, hence not cache friendly. Since AtA includes calls to Strassen for generic matrix multiplications, we also outline a time and space efficient implementation for this algorithm.

3.1 AtA in detail

Let $\mathbf{A} \in \mathbb{R}^{m \times n}$ be a rectangular matrix. The idea behind AtA is the following: at each recursive step, matrix $\mathbf{A}$ is divided into four

sub-matrices as follows:

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{1,1} & \mathbf{A}_{1,2} \\ \mathbf{A}_{2,1} & \mathbf{A}_{2,2} \end{bmatrix} \quad \begin{aligned} \mathbf{A}_{1,1} &= \mathbf{A}_{0:m_1,0:n_1} \in \mathbb{R}^{m_1 \times n_1} \\ \mathbf{A}_{1,2} &= \mathbf{A}_{0:m_1,n_1:n} \in \mathbb{R}^{m_1 \times n_2} \\ \mathbf{A}_{2,1} &= \mathbf{A}_{m_1:m,0:n_1} \in \mathbb{R}^{m_2 \times n_1} \\ \mathbf{A}_{2,1} &= \mathbf{A}_{m_1:m,n_1:n} \in \mathbb{R}^{m_2 \times n_2} \end{aligned} \quad (1)$$

being $m_1 := \lfloor \frac{m}{2} \rfloor$, $m_2 := \lceil \frac{m}{2} \rceil$, $n_1 := \lfloor \frac{n}{2} \rfloor$, $n_2 := \lceil \frac{n}{2} \rceil$. We address to sub-matrices of a matrix $\mathbf{A}$ as to indexed sub-blocks $(\mathbf{A}_{i,j})$ or with line and column intervals $(\mathbf{A}_{r_1:r_2,c_1:c_2})$. The product matrix $\mathbf{C} = \mathbf{A}^T \mathbf{A}$ is also split into four sub-matrices, resulting in the following:

$$\begin{aligned} \mathbf{C}_{1,1} &= \mathbf{A}_{1,1}^T \mathbf{A}_{1,1} + \mathbf{A}_{2,1}^T \mathbf{A}_{2,1} \in \mathbb{R}^{n_1 \times n_1}, \\ \mathbf{C}_{1,2} &= \mathbf{A}_{1,1}^T \mathbf{A}_{1,2} + \mathbf{A}_{2,1}^T \mathbf{A}_{2,2} \in \mathbb{R}^{n_1 \times n_2}, \\ \mathbf{C}_{2,1} &= \mathbf{A}_{1,2}^T \mathbf{A}_{1,1} + \mathbf{A}_{2,2}^T \mathbf{A}_{2,1} \in \mathbb{R}^{n_2 \times n_1}, \\ \mathbf{C}_{2,2} &= \mathbf{A}_{1,2}^T \mathbf{A}_{1,2} + \mathbf{A}_{2,2}^T \mathbf{A}_{2,2} \in \mathbb{R}^{n_2 \times n_2}. \end{aligned} \quad (2)$$

Both $\mathbf{C}_{1,1}$ and $\mathbf{C}_{2,2}$ consist of two addends that are, in turn, the left hand product of a matrix by its transpose. Hence, four recursive calls are employed to compute the sub-products $\mathbf{A}_{1,1}^T \mathbf{A}_{1,1}$ and $\mathbf{A}_{2,1}^T \mathbf{A}_{2,1}$ to obtain $\mathbf{C}_{1,1}$, and $\mathbf{A}_{1,2}^T \mathbf{A}_{1,2}$ and $\mathbf{A}_{2,2}^T \mathbf{A}_{2,2}$ to obtain $\mathbf{C}_{2,2}$.

Since for any matrix $\mathbf{A}$ the product $\mathbf{A}^T \mathbf{A}$ is symmetric, at each recursive step only the lower triangular part of the product matrix is computed, $\text{low}(\mathbf{C}_{i,i})$, $i = 1, 2$. As for component $\mathbf{C}_{2,1}$, in order to compute its two terms in the sum, we implement the generalized Strassen's algorithm for non-square matrices. The sub-matrix $\mathbf{C}_{1,2}$ is equal to $\mathbf{C}_{2,1}^T$, and therefore must not be explicitly computed. In Algorithm 1 we provide the pseudo-code of AtA. The base case occurs when the number of entries of the sub-matrix fits in the cache. In that case, the multiplication is performed by the BLAS function for $\mathbf{A}^T \mathbf{A}$, ?syrk, where the character ? represents a generic data type in accordance with standard notation used in manuals, [16]. In Algorithm 1, we also sketch our implementation of Strassen: before the actual recursive Strassen algorithm is called (STRASSEN), in FAST-STRASSEN we conveniently prepare an environment for memory efficiency by pre-allocating the memory for Strassen's algorithm, as explained in Section 3.3. The reduced number of multiplications in Strassen's algorithm is achieved by computing more matrix additions. In our implementation of STRASSEN, matrix additions are performed by calling the BLAS routine ?axpy (for the vector addition $\mathbf{y} = \alpha \mathbf{x} + \mathbf{y}$). The base-case condition in STRASSEN is analogous to the one of AtA. When the base-case condition holds, we call the BLAS routine ?gemm for the generic $\mathbf{A}^T \mathbf{B}$ multiplication. To handle odd-sized matrices, we do not implement well-known strategies such as peeling or padding, since these are known for introducing computational and memory overhead. Instead, we manage sums between matrices of discordant size by conveniently applying the BLAS routine ?axpy for array sums, so that it simulates padding of an extra 0 column or row, by excluding the last row and/or column of a sub-matrix from the sum.

AtA and FASTSTRASSEN are designed to be efficient alternatives to the BLAS routines ?gemm and ?syrk. Thus, they perform the same operations, respectively $\mathbf{C} = \alpha \mathbf{A}^T \mathbf{B} + \beta \mathbf{C}$ and $\mathbf{C} = \alpha \mathbf{A}^T \mathbf{A} + \beta \mathbf{C}$. However, we avoid introducing the scaling factor β from our algorithms for clarity of exposition, since $\mathbf{C}$ can be simply scaled before applying the algorithms.

Algorithm 1: AtA- Serial

Input: $\mathbf{A} \in \mathbb{R}^{m \times n}, \mathbf{C} \in \mathbb{R}^{n \times n}, \alpha \in \mathbb{R}$
Output: Lower triangular part of $\mathbf{C} = \alpha \mathbf{A}^T \cdot \mathbf{A} + \mathbf{C}$

1 **Procedure** AtA $(\mathbf{A}, \mathbf{C}, \alpha)$
2   **if** $m \times n \leq$ cache size **then**
3      $\mathbf{C} \leftarrow \mathbf{C}+$ blas_?syrk$(\mathbf{A}, \alpha)$;
4   return;
5   **else**
6   Initialize pointers to $\mathbf{A}_{i,j}$ and $\mathbf{C}_{i,j}$, $i, j = 1, 2$;
7   AtA $(\mathbf{A}_{1,1}, \mathbf{C}_{1,1}, \alpha)$;
8   AtA $(\mathbf{A}_{2,1}, \mathbf{C}_{1,1}, \alpha)$;
9   AtA $(\mathbf{A}_{1,2}, \mathbf{C}_{2,2}, \alpha)$;
10   AtA $(\mathbf{A}_{2,2}, \mathbf{C}_{2,2}, \alpha)$;
11   FASTSTRASSEN $(\mathbf{A}_{1,2}, \mathbf{A}_{1,1}, \mathbf{C}_{2,1}, \alpha)$;
12   FASTSTRASSEN $(\mathbf{A}_{2,2}, \mathbf{A}_{2,1}, \mathbf{C}_{2,1}, \alpha)$;
13 _____

Input: $\mathbf{A} \in \mathbb{R}^{m \times n}, \mathbf{B} \in \mathbb{R}^{m \times k}, \mathbf{C} \in \mathbb{R}^{n \times k}, \alpha \in \mathbb{R}$
Output: $\mathbf{C} = \alpha \mathbf{A}^T \cdot \mathbf{B} + \mathbf{C}$

14 **Procedure** FASTSTRASSEN$(\mathbf{A}, \mathbf{B}, \mathbf{C}, \alpha)$
15   Allocate $\mathbf{M} = \mathbf{0}^{n \times k/2}$;
16   Allocate $\mathbf{P} = \mathbf{0}^{m \times n/2}$;
17   Allocate $\mathbf{Q} = \mathbf{0}^{m \times k/2}$;
18   STRASSEN$(\mathbf{M}, \mathbf{P}, \mathbf{Q}, \mathbf{A}, \mathbf{B}, \mathbf{C}, \alpha)$;

3.2 Computational Complexity

The idea behind Strassen's algorithm is to perform a 2×2 matrix multiplication using 7 multiplications instead of 8, as required by naive matrix multiplication [33]. Nevertheless, Strassen's algorithm involves 18 sums between sub-matrices, thus leading to a computational complexity $T_S(n) \approx 7n^{\log_2 7}$.

In Algorithm 1, there are four recursive calls to AtA on basically halved dimensions, two calls to FASTSTRASSEN and 3 sums. Thus, we can derive the recurrence function for AtA runtime depending on the input size n as follows:

$$T(n) = 4T\left(\frac{n}{2}\right) + 2T_S\left(\frac{n}{2}\right) + 3\left(\frac{n}{2}\right)^2 \approx \frac{2}{3}T_S(n). \quad (3)$$

The overall computational complexity of AtA reduces the one of the general matrix multiplication $\mathbf{A}^T \mathbf{A}$, amounting to $n^2(n + 1)$, and of Strassen's algorithm naively applied for computing $\mathbf{A}^T \mathbf{A}$, that would require the same number of products as for the general matrix multiplication, and only 16 sums instead of the 18 matrix additions in the original Strassen's formulation.

3.3 Space complexity

In AtA, at each recursive step, pointers to the current portions of $\mathbf{A}$ and $\mathbf{C}$ are initialized so that, when the condition for the base-case occurs, the matrix multiplications are carried out on the correct sub-matrices of $\mathbf{A}$, and stored in the corresponding locations in $\mathbf{C}$.

Strassen's algorithm for general matrix multiplication is called twice. One drawback of the naive Strassen implementation is the great amount of memory allocated at each recursive step to store the results of the intermediate matrix additions required by the algorithm. In order to avoid frequent memory allocations and releases,

we call recursive Strassen (STRASSEN) on pre-allocated matrices, $\mathbf{M}$, $\mathbf{P}$ and $\mathbf{Q}$ (FASTSTRASSEN). The size of such matrices is sufficiently large to store all intermediate matrix operation results throughout the recursive calls. In fact, given an $n \times n$ matrix, at each recursive step we halve both the dimensions, rounding up the result to the nearest integer when matrices have odd sizes. By doing so, the amount of memory used by the algorithm when the base case is reached if

$$\sum_{i=1}^{\log_2 n} \frac{(n + \log_2 n)^2}{4^i} = (n + \log_2 n)^2 \left(\frac{1}{3} - \frac{4}{3n^2} \right) \le \frac{n^2}{2} \quad (4)$$

which, multiplied by the three supporting matrices $\mathbf{M}$, $\mathbf{P}$ and $\mathbf{Q}$, results in a total of $\frac{3}{2}n^2$. Although the overall space complexity of Strassen does not change, we are able to save time for memory allocation at each recursive step. Consequently, the space complexity of ATA is $S(n) = \frac{3}{2}n^2$.

In Section 5, we show that Strassen's algorithm benefits from the described strategy for memory allocation.

3.4 Cache Complexity

In this section, we show the cache complexity of ATA. We assume the ideal cache model and we denote with M the cache size, and with b the size of the cache line.

PROPOSITION 3.1. *The cache complexity of AtA, $C_{AtA}(n; M, b)$, is the same as the cache complexity of Strassen, $C_S(n; M, b) = \Theta(1 + n^2/b + n^{\log_2(7)}/b\sqrt{M})$, [17].*

Algorithm 2: RECURSIVEGEMM

Input: $\mathbf{A} \in \mathbb{R}^{m \times n}, \mathbf{B} \in \mathbb{R}^{m \times k}, \mathbf{C} = \mathbf{0}^{n \times k}$
Output: $\mathbf{C} = \mathbf{A}^T \cdot \mathbf{B}$
1 **Procedure** RECURSIVEGEMM($\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$)
2   **if** $m \times n + m \times k \le$ cache size **then**
3     $\mathbf{C} += $ blas_?gemm($\mathbf{A}^T$, $\mathbf{B}$);
4   return;
5   **for** $i = 1, 2$ **do**
6   **for** $j = 1, 2$ **do**
7   **for** $k = 1, 2$ **do**
8   RECURSIVEGEMM($\mathbf{A}_{k,i}$, $\mathbf{B}_{k,j}$, $\mathbf{C}_{i,j}$);

PROOF. We prove the thesis by induction. First, we observe that $C_{AtA}(2; M, b) = 6C_S(1; M, b) \le 7C_S(1; M, b) = C_S(2; M, b)$. Assuming as inductive hypothesis that $C_{AtA}(n/2; M, b) \le C_S(n/2; M, b)$, it holds that:

$$C_{AtA}(n; M, b) = 4C_{AtA}(n/2; M, b) + 2C_S(n/2; M, b)$$
$$\le 6C_S(n/2; M, b) \le 7C_S(n/2; M, b) = C_S(n; M, b).$$

Furthermore, notice that: $C_S(n/2; M, b) \le C_{AtA}(n; M, b) \le C_S(n; M, b)$. Hence, the thesis holds. □

4 PARALLEL ATA

Our algorithm for the $\mathbf{A}^T\mathbf{A}$ product, AtA, can be conveniently parallelized to work on both shared and distributed-memory systems. We will refer to our shared and distributed-memory algorithms

for $\mathbf{A}^T\mathbf{A}$ as AtA-S and AtA-D, respectively. Our parallel implementations of AtA take advantage of the recursive nature of AtA to distribute tasks (and possibly data) to different processes in an efficient way. To do so, an initial phase that implements a scheduler covering the recursion tree of AtA is integrated in both parallel algorithms. In this way, we assign a task to each different parallel process, as we explain in Section 4.1. After this preliminary phase, each process knows which sub-problem it has to solve.

4.1 Preliminary phase: task assignment

Usually, recursive algorithms are parallelized with a fork-join paradigm, according to their natural behaviour: at each recursive call, a new thread is created to accomplish that call. However, repeatedly creating and killing threads introduces a significant overhead, especially when it happens as a nested procedure. A parallelized for loop approach can usually improve this thread start-up overhead. For this reason, rather than addressing the problem by distributing recursive calls between newly created threads, we simulate the behaviour of a fork-join algorithm to determine, for each thread, on which sub-matrices it must work. This is particularly useful to generalize our approach to both shared memory and distributed settings.

4.1.1 Building the task tree. To conveniently distribute tasks among P parallel processes collaborating to compute $\mathbf{A}^T\mathbf{A}$, in the first phase of our algorithms, each process builds the recursion tree of a modified version of AtA, that we shall call AtANAIVE, and explores a part of it with a breadth-first search (BFS), see Figure 1. AtANAIVE considers classic recursive general matrix multiplication instead of Strassen, and can be easily implemented by modifying Algorithm 1 to call RECURSIVEGEMM instead of STRASSEN. RECURSIVEGEMM, summarized in Algorithm 2, is a recursive algorithm for the naive general matrix multiplication. The reasons behind this choice will be explained in Section 4.1.3. We define the *task tree*, denoted with $\mathcal{T}$, to be the sub-tree of the recursion tree of AtA, obtained by spanning the latter with a BFS, that is interrupted as soon as $\mathcal{T}$ counts P leaves, labeled from 0 to $P - 1$. Both AtA-S and AtA-D implement the task tree, but with some differences concerning data and task division. In AtA-D, each p-th leaf corresponds to the task that process p has to fulfil, and contains directives on both the computational and communication activity that is due to the corresponding process. Specifically, a leaf task t provides the following information:

(1) *t.computationType*: Which type of computation process p has to carry out. It can be either a $\mathbf{A}^T\mathbf{A}$ or a $\mathbf{A}^T\mathbf{B}$ multiplication;

(2) *t.X.offset* and *t.X.q*, with $\mathbf{X} \in \{\mathbf{A}, \mathbf{B}, \mathbf{C}\}$, $q \in \{m, n\}$: The row and column offsets as well as the size of the sub-matrices of $\mathbf{A}$ and $\mathbf{C}$ process p has to work on;

(3) *t.parent*: The parent process that sends sub-matrices of $\mathbf{A}$ to its children (during the distribution phase), and to which process p has to send the result of the task that was assigned to it or, if p is the parent, the information on its children' tasks (during result retrieval).

Inner nodes of $\mathcal{T}$ instead, represent tasks concerning data distribution and retrieval, possibly involving sums of sub-matrices of

Figure 1 tree (root $p_0 : \mathbf{A}^T\mathbf{A}(\mathbf{C} = \mathbf{A}^T\mathbf{A})$):

Level 2 nodes:
- $p_0 : \mathbf{A}^T\mathbf{B}(\mathbf{C}_{\frac{n}{2}:n,0:\frac{n}{2}} = \mathbf{A}^T_{0:\frac{n}{2},\frac{n}{2}:n}\mathbf{A}_{0:\frac{n}{2},0:\frac{n}{2}})$
- $p_1 : \mathbf{A}^T\mathbf{B}(\mathbf{C}_{\frac{n}{2}:n,0:\frac{n}{2}} = \mathbf{A}^T_{\frac{n}{2}:n,\frac{n}{2}:n}\mathbf{A}_{\frac{n}{2}:n,0:\frac{n}{2}})$
- $p_2 : \mathbf{A}^T\mathbf{A}(\mathbf{C}_{0:\frac{n}{2},0:\frac{n}{2}} = \mathbf{A}^T_{0:\frac{n}{2},0:\frac{n}{2}}\mathbf{A}_{0:\frac{n}{2},0:\frac{n}{2}})$
- $p_3 : \mathbf{A}^T\mathbf{A}(\mathbf{C}_{0:\frac{n}{2},0:\frac{n}{2}} = \mathbf{A}^T_{\frac{n}{2}:n,0:\frac{n}{2}}\mathbf{A}_{\frac{n}{2}:n,0:\frac{n}{2}})$
- $p_4 : \mathbf{A}^T\mathbf{A}(\mathbf{C}_{\frac{n}{2}:n,\frac{n}{2}:n} = \mathbf{A}^T_{0:\frac{n}{2},\frac{n}{2}:n}\mathbf{A}_{0:\frac{n}{2},\frac{n}{2}:n})$
- $p_5 : \mathbf{A}^T\mathbf{A}(\mathbf{C}_{\frac{n}{2}:n,\frac{n}{2}:n} = \mathbf{A}^T_{\frac{n}{2}:n,\frac{n}{2}:n}\mathbf{A}_{\frac{n}{2}:n,\frac{n}{2}:n})$

Level-3 (leaf) nodes and corresponding boxed AtA-S leaves:

- $p_0 : \mathbf{A}^T\mathbf{B}(\mathbf{C}_{\frac{n}{2}:\frac{3n}{4},0:\frac{n}{4}} = \mathbf{A}^T_{0:\frac{n}{2},\frac{n}{2}:\frac{3n}{4}}\mathbf{A}_{0:\frac{n}{2},0:\frac{n}{4}})$ — $\boxed{\mathbf{A}^T\mathbf{B}(\mathbf{C}_{\frac{n}{2}:\frac{3}{4}n,0:\frac{n}{8}} = \mathbf{A}^T_{0:n,\frac{n}{2}:\frac{3}{4}n}\mathbf{A}_{0:n,0:\frac{n}{8}})}$
- $p_6 : \mathbf{A}^T\mathbf{B}(\mathbf{C}_{\frac{3n}{4}:n,0:\frac{n}{4}} = \mathbf{A}^T_{0:\frac{n}{2},\frac{3n}{4}:n}\mathbf{A}_{0:\frac{n}{2},0:\frac{n}{4}})$ — $\boxed{\mathbf{A}^T\mathbf{A}(\mathbf{C}_{0:\frac{n}{2},0:\frac{n}{4}} = \mathbf{A}^T_{0:n,0:\frac{n}{2}}\mathbf{A}_{0:n,0:\frac{n}{4}})}$
- $p_7 : \mathbf{A}^T\mathbf{B}(\mathbf{C}_{\frac{n}{2}:\frac{3n}{4},\frac{n}{4}:\frac{n}{2}} = \mathbf{A}^T_{0:\frac{n}{2},\frac{n}{2}:\frac{3n}{4}}\mathbf{A}_{0:\frac{n}{2},\frac{n}{4}:\frac{n}{2}})$ — $\boxed{\mathbf{A}^T\mathbf{A}(\mathbf{C}_{\frac{n}{4}:\frac{n}{2},\frac{n}{4}:\frac{n}{2}} = \mathbf{A}^T_{0:n,\frac{n}{4}:\frac{n}{2}}\mathbf{A}_{0:n,\frac{n}{4}:\frac{n}{2}})}$
- $p_8 : \mathbf{A}^T\mathbf{B}(\mathbf{C}_{\frac{3n}{4}:n,\frac{n}{4}:\frac{n}{2}} = \mathbf{A}^T_{0:\frac{n}{2},\frac{3n}{4}:n}\mathbf{A}_{0:\frac{n}{2},\frac{n}{4}:\frac{n}{2}})$ — $\boxed{\mathbf{A}^T\mathbf{A}(\mathbf{C}_{\frac{n}{2}:\frac{3}{4}n,\frac{n}{2}:\frac{3}{4}n} = \mathbf{A}^T_{0:n,\frac{n}{2}:\frac{3}{4}n}\mathbf{A}_{0:n,\frac{n}{2}:\frac{3}{4}n})}$
- $p_1 : \mathbf{A}^T\mathbf{B}(\mathbf{C}_{\frac{n}{2}:\frac{3n}{4},0:\frac{n}{4}} = \mathbf{A}^T_{\frac{n}{2}:n,\frac{n}{2}:\frac{3n}{4}}\mathbf{A}_{\frac{n}{2}:n,0:\frac{n}{4}})$ — $\boxed{\mathbf{A}^T\mathbf{B}(\mathbf{C}_{\frac{n}{4}:n,0:\frac{n}{8}} = \mathbf{A}^T_{0:n,\frac{n}{4}:n}\mathbf{A}_{0:n,0:\frac{n}{8}})}$
- $p_9 : \mathbf{A}^T\mathbf{B}(\mathbf{C}_{\frac{3n}{4}:n,0:\frac{n}{4}} = \mathbf{A}^T_{\frac{n}{2}:n,\frac{3n}{4}:n}\mathbf{A}_{\frac{n}{2}:n,0:\frac{n}{4}})$ — $\boxed{\mathbf{A}^T\mathbf{A}(\mathbf{C}_{\frac{3}{4}n:n,\frac{3}{4}n:n} = \mathbf{A}^T_{0:n,\frac{3}{4}n:n}\mathbf{A}_{0:n,\frac{3}{4}n:n})}$
- $p_{10} : \mathbf{A}^T\mathbf{B}(\mathbf{C}_{\frac{n}{2}:\frac{3n}{4},\frac{n}{4}:\frac{n}{2}} = \mathbf{A}^T_{\frac{n}{2}:n,\frac{n}{2}:\frac{3n}{4}}\mathbf{A}_{\frac{n}{2}:n,\frac{n}{4}:\frac{n}{2}})$ — $\boxed{\mathbf{A}^T\mathbf{B}(\mathbf{C}_{\frac{n}{2}:\frac{3}{4}n,\frac{n}{4}:\frac{n}{4}} = \mathbf{A}^T_{0:n,\frac{n}{2}:\frac{3}{4}n}\mathbf{A}_{0:n,\frac{n}{8}:\frac{n}{4}})}$
- $p_{11} : \mathbf{A}^T\mathbf{B}(\mathbf{C}_{\frac{3n}{4}:n,\frac{n}{4}:\frac{n}{2}} = \mathbf{A}^T_{\frac{n}{2}:n,\frac{3n}{4}:n}\mathbf{A}_{\frac{n}{2}:n,\frac{n}{4}:\frac{n}{2}})$ — $\boxed{\mathbf{A}^T\mathbf{B}(\mathbf{C}_{\frac{3}{4}n:n,\frac{n}{8}:\frac{n}{4}} = \mathbf{A}^T_{0:n,\frac{3}{4}n:n}\mathbf{A}_{0:n,\frac{n}{8}:\frac{n}{4}})}$
- $p_2 : \mathbf{A}^T\mathbf{A}(\mathbf{C}_{0:\frac{n}{2},0:\frac{n}{2}} = \mathbf{A}^T_{0:\frac{n}{4},0:\frac{n}{2}}\mathbf{A}_{0:\frac{n}{2},0:\frac{n}{2}})$ — $\boxed{\mathbf{A}^T\mathbf{B}(\mathbf{C}_{\frac{3}{4}n:n,\frac{n}{2}:\frac{5}{8}n} = \mathbf{A}^T_{0:n,\frac{3}{4}n:n}\mathbf{A}_{0:n,\frac{n}{2}:\frac{5}{8}n})}$
- $p_{12} : \mathbf{A}^T\mathbf{A}(\mathbf{C}_{0:\frac{n}{2},0:\frac{n}{2}} = \mathbf{A}^T_{\frac{n}{4}:\frac{n}{2},0:\frac{n}{2}}\mathbf{A}_{\frac{n}{4}:\frac{n}{2},0:\frac{n}{2}})$ — $\boxed{\mathbf{A}^T\mathbf{B}(\mathbf{C}_{\frac{n}{2}:\frac{3}{4}n,\frac{n}{8}:\frac{n}{2}} = \mathbf{A}^T_{0:n,\frac{n}{2}:\frac{3}{4}n}\mathbf{A}_{0:n,\frac{n}{8}:\frac{n}{2}})}$
- $p_3 : \mathbf{A}^T\mathbf{A}(\mathbf{C}_{0:\frac{n}{2},0:\frac{n}{2}} = \mathbf{A}^T_{\frac{n}{2}:\frac{3n}{4},0:\frac{n}{2}}\mathbf{A}_{\frac{n}{2}:\frac{3n}{4},0:\frac{n}{2}})$ — $\boxed{\mathbf{A}^T\mathbf{B}(\mathbf{C}_{\frac{3}{4}n:n,0:\frac{n}{8}} = \mathbf{A}^T_{0:n,\frac{3}{4}n:n}\mathbf{A}_{0:n,0:\frac{n}{8}})}$
- $p_{13} : \mathbf{A}^T\mathbf{A}(\mathbf{C}_{0:\frac{n}{2},0:\frac{n}{2}} = \mathbf{A}^T_{\frac{3n}{4}:n,0:\frac{n}{2}}\mathbf{A}_{\frac{3n}{4}:n,0:\frac{n}{2}})$ — $\boxed{\mathbf{A}^T\mathbf{B}(\mathbf{C}_{\frac{3}{4}n:n,\frac{n}{8}:\frac{n}{2}} = \mathbf{A}^T_{0:n,\frac{3}{4}n:n}\mathbf{A}_{0:n,\frac{n}{8}:\frac{n}{2}})}$
- $p_4 : \mathbf{A}^T\mathbf{A}(\mathbf{C}_{\frac{n}{2}:n,\frac{n}{2}:n} = \mathbf{A}^T_{0:\frac{n}{2},\frac{n}{2}:n}\mathbf{A}_{0:\frac{n}{4},\frac{n}{2}:n})$ — $\boxed{\mathbf{A}^T\mathbf{B}(\mathbf{C}_{\frac{n}{2}:\frac{3}{4}n,\frac{n}{4}:\frac{3}{8}n} = \mathbf{A}^T_{0:n,\frac{n}{2}:\frac{3}{4}n}\mathbf{A}_{0:n,\frac{n}{4}:\frac{3}{8}n})}$
- $p_{14} : \mathbf{A}^T\mathbf{A}(\mathbf{C}_{\frac{n}{2}:n,\frac{n}{2}:n} = \mathbf{A}^T_{\frac{n}{4}:\frac{n}{2},\frac{n}{2}:n}\mathbf{A}_{\frac{n}{4}:\frac{n}{2},\frac{n}{2}:n})$ — $\boxed{\mathbf{A}^T\mathbf{B}(\mathbf{C}_{\frac{n}{4}:\frac{n}{2},\frac{n}{8}:\frac{n}{4}} = \mathbf{A}^T_{0:n,\frac{n}{4}:\frac{n}{2}}\mathbf{A}_{0:n,\frac{n}{8}:\frac{n}{4}})}$
- $p_5 : \mathbf{A}^T\mathbf{A}(\mathbf{C}_{\frac{n}{2}:n,\frac{n}{2}:n} = \mathbf{A}^T_{\frac{n}{2}:n,\frac{3n}{4},\frac{n}{2}:n}\mathbf{A}_{\frac{n}{2}:n,\frac{3n}{4}:n})$ — $\boxed{\mathbf{A}^T\mathbf{B}(\mathbf{C}_{\frac{3}{4}n:n,\frac{n}{4}:\frac{3}{8}n} = \mathbf{A}^T_{0:n,\frac{3}{4}n:n}\mathbf{A}_{0:n,\frac{n}{4}:\frac{3}{8}n})}$
- $p_{15} : \mathbf{A}^T\mathbf{A}(\mathbf{C}_{\frac{n}{2}:n,\frac{n}{2}:n} = \mathbf{A}^T_{\frac{3n}{4}:n,\frac{n}{2}:n}\mathbf{A}_{\frac{3n}{4}:n,\frac{n}{2}:n})$ — $\boxed{\mathbf{A}^T\mathbf{B}(\mathbf{C}_{\frac{3}{4}n:n,\frac{5}{8}n:\frac{3}{4}n} = \mathbf{A}^T_{0:n,\frac{3}{4}n:n}\mathbf{A}_{0:n,\frac{5}{8}n:\frac{3}{4}n})}$

Figure 1: A tree of 16 processes distributing $A \in \mathbb{R}^{n \times n}$. Boxed labels on the right-hand side are the leaf nodes of the tree generated by AtA-S, corresponding to computation tasks assigned to corresponding processes in the left-hand side leaf labels.

$\mathbf{C} = \mathbf{A}^T\mathbf{A}$, and consequent communication (point 3 of the previous list), and are executed by a subset of processes. In contrast, in AtA-S only leaf nodes of $\mathcal{T}$ correspond to a task, whereas inner nodes are ignored, as no communication is involved. For the same reason, leaf tasks only include information about what kind of computation the corresponding threads have to carry out and on the sub-matrices they have to work on (points 1 and 2 of the previous list).

4.1.2 Load Balancing. The task tree of AtA-D is created so that, at each level, given P available processes, $\alpha \cdot P$ processes compute a general $\mathbf{A}^T\mathbf{B}$ matrix multiplication; for the remaining $(1 - \alpha) \cdot P$ processes, a task for a $\mathbf{A}^T\mathbf{A}$ multiplication is assigned to them. Here, $\alpha \in (0, 1)$ is a parameter for balancing the workload among distributed processes, as the computational complexity of a $\mathbf{A}^T\mathbf{A}$ product is lower than the one of $\mathbf{A}^T\mathbf{B}$. The task tree $\mathcal{T}$ is built by calling RecursiveGEMM (whose computational complexity is roughly twice the one of AtA, $T(n)$). Therefore the number of multiplications carried out in $\mathcal{T}$ to perform $\mathbf{A}^T\mathbf{B}$ is twice the one needed to compute $\mathbf{A}^T\mathbf{A}$. The load balancing parameter must be such that $4 \cdot T(n)/(1-\alpha)P = 2 \cdot 2T(n)/\alpha P$. In accordance, we set $\alpha = 1/2$. This task division is repeated recursively at each level, by progressively decreasing the number of available processes, P. The number of recursive parallel steps depends on P and α. In particular, for $\alpha = 0.5$, the number of parallel levels in the task tree, ℓ is given by the following expression:

$$\ell(P = 1) = 0, \quad \ell(2 \le P \le 6) = 1$$
$$\ell(P > 6) = 1 + k + \text{sign}\left(\frac{P}{4} \mod 8^{\max\{k;1\}}\right), \tag{5}$$

where $k = \max\left\{k \in \mathbb{N} : \frac{P/4}{8^k} \ge 1\right\}$ and $\text{sign}(x)$ is the sign function, returning 0 for $x = 0$ and 1 for $x > 0$. Indeed, when AtA-D is

called on P processes, $P/2$ of them are going to compute $\mathbf{C}_{2,1}$; out of them, $P/4$ processes compute $\mathbf{A}^T_{1,2}\mathbf{A}_{1,1}$, whereas the remaining $P/4$ are in charge for $\mathbf{A}^T_{2,2}\mathbf{A}_{2,1}$ (see Equation 2). These tasks are in turn distributed among 8 processes each, recursively (corresponding to the eight recursive calls of RecursiveGEMM). This splitting is repeated as long as it possible (i.e., until $P/4/8^k \ge 1$). If by doing so, all $P/4$ processes are used (i.e., $P/4$ is a multiple of 8^k, for some k), all processes work on equally sized matrices. Otherwise, some processes will further split their tasks to smaller matrices, resulting in an additional parallel level. We say that the last parallel level is *complete* when all leaves corresponding to $\mathbf{A}^T\mathbf{A}$ tasks are grouped in bunches of 6 siblings, and when all leaves corresponding to $\mathbf{A}^T\mathbf{B}$ tasks are grouped in bunches of 8 siblings.

The task tree for AtA-S is quite different. In order to avoid concurrent overlapping writes, input matrices are tiled in horizontal and vertical blocks, as depicted in Figure 2. This way, we ensure that each thread computes a different $\mathbf{C}_{i,j}$. With this new scheme, we make three recursive calls to AtA (instead of 6) and four recursive calls to FastStrassen (instead of 8). Therefore, the number of parallel levels in AtA-S, given P threads, is the following:

$$\ell(P = 1) = 0, \quad \ell(P = 2, 3) = 1,$$
$$\ell(P > 3) = 1 + k + \text{sign}\left(\frac{P}{2} \mod 4^{\max\{k;1\}}\right), \tag{6}$$

with $k = \max\left\{k \in \mathbb{N} : \frac{P/2}{4^k} \ge 1\right\}$. In Figure 1, we show an example of the task tree with 16 processes for AtA-D, and the leaf nodes of the task tree for AtA-S (boxed).

4.1.3 Naive matrix multiplication over Strassen. In our parallel algorithms, we do not rely on Strassen for general $\mathbf{A}^T\mathbf{B}$ matrix multiplication when building the recursion tree, that instead is created by simulating AtANaive. This is done with the goal of optimizing the resources of distributed architectures, as the naive general matrix-multiplication algorithm does not allocate the additional memory required by Strassen, resulting in a faster memory management. Furthermore, Strassen's algorithm would possibly cause a hardly manageable workload unbalance between processes implementing an $\mathbf{A}^T\mathbf{A}$ multiplication, and those that would be in charge of computing the intermediate matrix sums appearing in Strassen's algorithm. However, Strassen's algorithm can still be used at leaf-level computation.

4.2 Shared-memory AtA

AtA can be implemented with a shared-memory parallel paradigm on multi-core machines. We rely on OpenMP to efficiently distribute the workload between threads. Each thread simulates the recursion of AtANaive as described in Section 4.1. The workload is distributed so that each thread writes in a different memory location, hence there is no need of handling data collisions of any kind. Instead, the problem is divided in a fashion that makes it embarrassingly parallel. We call AtA-S our multi-threaded algorithm for $\mathbf{A}^T\mathbf{A}$.

4.2.1 AtA-S in detail. Let us denote with P the number of available threads. Our algorithm for multi-threaded machines, AtA-S, can be divided into two phases. During the first phase, one task is assigned to each thread by simulating the recursion of AtANaive, as described in Section 4.1. In order to prevent memory collisions and to achieve embarrassing parallelism, tasks are organized so that each thread writes on a different and disjoint memory location. This is done by dividing the resulting matrix $\mathbf{C}$ into four blocks, as shown in Equation 2, whereas $\mathbf{A}$ is tiled vertically or horizontally, instead of in 2×2 blocks (see Figure 2). This procedure avoids concurrent writing management, it guarantees data and thread reuse and relies on the equality:

$$\mathbf{C}_{i,j} = \mathbf{A}_{i,1}\mathbf{B}_{1,j} + \mathbf{A}_{i,2}\mathbf{B}_{2,j} = \mathbf{A}_{i,*}\mathbf{B}_{*,j}, \tag{7}$$

for $i, j = 1, 2$. Such instruction and data assignment allows for a faster execution, since threads never need to synchronize.

During the second phase of AtA-S, each thread retrieves its task from the tree $\mathcal{T}$, specifying which routine (either AtA or FastStrassen) the corresponding thread must call, and on which sub-matrices of $\mathbf{A}$ and $\mathbf{C}$ it must operate. On multicore systems, this means that data reuse in both L1 and L2 cache is optimized, since each thread operates on the same data throughout its entire lifespan. Since the tasks correspond to disjoint sub-problems, at the end of the computation each thread only needs to synchronize with the others, then the algorithm stops. In Algorithm 3 we provide the pseudo-code of AtA-S.

4.2.2 Computational Complexity of AtA-S. We study the time complexity $T(n, P)$ of AtA-S to perform the multiplication $\mathbf{A}^T\mathbf{A}$ on an $n \times n$ matrix $\mathbf{A}$ and distributing the workload between P processes.

At first, the algorithm needs to generate the task tree and each process has to retrieve its task. These procedures have the same complexity as a BFS visit on a tree with P leaves, hence $O(P)$.

Algorithm 3: AtA-S- Shared

Input: $\mathbf{A} \in \mathbb{R}^{m \times n}$
Output: Lower triangular part of $\mathbf{C} = \mathbf{A}^T \cdot \mathbf{A}$

1 **Procedure** AtA-S($\mathbf{A}$)
2   Generate tree $\mathcal{T}$;
3   **parfor** each leaf-node v of $\mathcal{T}$ **do**
4   Get task t from node v;
5   **if** $t.computationType = \mathbf{A}^T\mathbf{A}$ **then**
6   AtA ($\mathbf{A}_{t.\mathbf{A}.offset}$, $\mathbf{C}_{t.\mathbf{C}.offset}$, 1);
7   **else if** $t.computationType = \mathbf{A}^T\mathbf{B}$ **then**
8   FastStrassen ($\mathbf{A}_{t.\mathbf{A}.offset}$, $\mathbf{A}_{t.\mathbf{B}.offset}$, $\mathbf{C}_{t.\mathbf{C}.offset}$, 1);

The time complexity of the second step corresponds to the one of the most expensive leaf task, which appears at the end of a path of RecursiveGEMM calls. At level l, the size of the product matrix $\mathbf{C}$ is reduced to a block of size $n/2^l \times n/2^l$, resulting from a multiplication between $n/2^l \times n$ and $n \times n/2^l$ matrices. Thus, the total complexity is reduced by $4^{\ell(P)}$, being $\ell(P)$ the number of levels in the task tree. Hence the total complexity of the algorithm is:

$$T(n, P) = O(P) + O\left(\frac{1}{4^{\ell(P)}} n^{\log_2 7}\right). \tag{8}$$

Notice that $\ell(P)$ is a discrete, non-injective function. Hence, especially with few processes, the speed-up behaves like a step function. Despite this behaviour, $\ell(P) \approx \log_4 P$, meaning with large numbers of processes we achieve a theoretical linear speed-up.

Figure 2: Multiplication with vertical/horizontal tiling.

4.3 Distributed-memory AtA

Modern computers are equipped with an ever-increasing number of cores inside CPU chips. However, when it comes to massive volumes of data, computationally intensive tasks such as matrix multiplication are simply prohibitive, even for the most recent 16- or 32-cores chipsets, and even with hyper-threading capabilities. Distributed parallelism plays a crucial role in this setting, as it allows to distribute the workload between multiple machines. In such an environment, providing fast distributed algorithms for operations in Linear Algebra, including $\mathbf{A}^T\mathbf{A}$ multiplication, is a key task to limit bottlenecks.

In this section, we describe a distributed algorithm for $\mathbf{A}^T\mathbf{A}$, that works for any matrix size and with arbitrarily many processes and cores. We shall refer to this algorithm as AtA-D. AtA-D follows a distribute-compute-retrieve paradigm, as initially the input matrix $\mathbf{A}$ is stored on the root process only, and distributed to other processes according to their tasks. Finally, the resulting matrix $\mathbf{C} = \mathbf{A}^T\mathbf{A}$ is retrieved back by the root process. We implement a parallel communication scheme to limit data transfer overhead.

Algorithm 4: ATA-D- Distributed

Input: $A \in \mathbb{R}^{m \times n}$
Output: Lower triangular part of $C = A^T \cdot A$

1 **Procedure** ATA-D(A)
2   Generate tree $\mathcal{T}$;
3   **for** each v of $\mathcal{T}$ in the path from my leaf to the root **do**
4   Get my task t from node v;
5   **if** v is a leaf **then**
6   **if** $t.computationType = A^TA$ **then**
7          $C_{t.C.offset} = A^TA(A_{t.A.offset})$;
8   **else if** $t.computationType = A^TB$ **then**
9          $C_{t.C.offset} = A^TB(A_{t.A.offset}, A_{t.B.offset})$;
10   **if** $t.parent \neq$ my ID **then**
11   Send $C_{t.C.offset}$ to $t.parent$;
12   **else**
13   Receive $C_{children.t.C.offset}$ from my children;
14   Sum over the sub-matrices and store result in C;

4.3.1 ATA-D in detail. Let P be the number of distributed processes. In ATA-D, each process p first builds the task tree $\mathcal{T}$ as described in Section 4.1. To understand in detail how $\mathcal{T}$ is used in ATA-D, we shall refer to the example of Figure 1. As we said, each node represents a task, but only tasks contained in leaf nodes correspond to an actual matrix multiplication. Inner nodes instead represent tasks assigned only to the parents of the nodes branching out of them, and they are necessary to retrieve and combine the portions of the result matrix scattered among different processes, and eventually to send them, level by level, up to the root process, p_0. In the example of Figure 1, $\mathcal{T}$ is the task tree for $P = 16$ processes on a square matrix. Leaf nodes are generated so that processes p_0, p_1 and $p_6 \ldots, p_{11}$ share the workload to compute $C_{2,1}$. The remaining half of the processes is devoted to compute $C_{1,1}$ and $C_{2,2}$. If the number of distributed processes is not enough to make a complete level, as in this example, instead of calling multiple tasks on different tiles of the matrices, processes perform either an A^TA or a A^TB operation on vertically and horizontally tiled sub-matrices at the leaf-level. For instance, observe the first batch of sibling-leaves in Figure 1. To compute $C_{n/2:n,0:n/2} = A^T_{0:n/2,n/2:n}A_{0:n/2,0:n/2}$, ATANAIVE would perform 8 recursive calls to A^TB; in ATA-D, each of these calls is served by one distributed process, if available. When this is not the case, as in the example that we are considering, processes p_0, p_6, p_7, p_8 divide $A_{0:n/2,n/2:n}$ and $A_{0:n/2,0:n/2}$ in vertical tiles so as to compute the related portions of C as depicted in Figure 2. When the computation is over, partial results are collected by the parents of each group of siblings (processes p_i, $i = 0, \ldots, 5$). This operation is iterated by traversing the tree up to its root, p_0, and allows for a convenient parallel communication reducing data transfer overhead. In order to optimize the communication and to reduce the exchanged data volume, we encode the sub-matrices resulting from A^TA operations as packed lower triangular matrices. Nevertheless, the entire operation, once it returns to the root process, still produces a standard square matrix. In Algorithm 4, we provide the pseudocode of ATA-D. In line 11, if the process has to fulfill a A^TB task, it sends to its parent the entire sub-matrix $C_{t.C.offset}$;

otherwise, it only sends $\text{low}(C_{t.C.offset})$. In lines 7 and 9, A^TA and A^TB may refer to ATA or blas_?syrk, and to FASTSTRASSEN or blas_?gemm, respectively. As we shall see in Section 5, the real benefit of using our implementation of ATA and FASTSTRASSEN arises on matrices with larger size, therefore they are favourable when handling larger volumes of data.

4.3.2 Computational and Communication Complexity of ATA-D. In contrast to parallel algorithms for distributed matrices, ATA-D does not include any communication between processes at computation time, as the input matrix is scattered among distributed processes so that they own the exact portions of A on which they have to operate.

PROPOSITION 4.1. *The computational cost of ATA-D (Algorithm 4) on a matrix of size n and with using P processes, $C(n, P)$ is:*

$$C(n, P) = O\left((n/2^{\ell(P)})^2 \cdot n/2^{\ell(P)-1}\right),$$

if the load balancing parameter α is set to 0.5.

PROOF. $C(n, P)$ depends on the number of recursive levels that can be layered with the available resources and on α. For $\alpha = 0.5$, the computational complexity of ATA-D is given by the time for computing A^TB on matrices of size at most $n/2^{\ell(P)} \times n/2^{\ell(P)-1}$, that is $O\left((n/2^{\ell(P)})^2 \cdot n/2^{\ell(P)-1}\right)$, where $\ell(P)$ is the number of parallel levels defined in Equation 5. □

We express the communication cost for matrix distribution and result retrieval in terms of latency and bandwidth costs of a distributed algorithm, denoted with $L(n, P)$ and $BW(n, P)$, respectively, using the same definitions introduced in [3] and adopted also in [26]. Latency cost is the communicated-message count, whereas bandwidth is expressed in terms of communicated-word count. Messages and words counts are computed along the critical path of the distributed algorithm, as defined in [37].

PROPOSITION 4.2. *The latency of ATA-D on a matrix of size $n \times n$ and with P processes is $L(n, P) = O(2[7 \cdot (\ell(P)-1)+5])$. Its bandwidth is $BW(n, P) \leq 6(n/2)^2 + \frac{n(n+2)}{2} + 7/6n^2(1 - 1/4^{\ell(P)-2})$.*

PROOF. In ATA-D, the critical path corresponds to the sequence of communication operations carried out by the root process p_0. After the first parallel level, p_0 works on a A^TB task and shares its workload with 7 other processes at each parallel level. When the compute phase is over, at each level $l \in \{2, \ldots, \ell(P)\}$ process p_0 collects partial results from its (at most) seven children; at level $l = 1$, it retrieves the entire matrix $C = A^TA$ by combining together the results of its five siblings. This operation is carried out both for data distribution and result collection. Hence, $L(n, P) = O(2[7 \cdot (\ell(P) - 1) + 5])$.
During the data distribution phase, message sizes (i.e., portions of input matrix A) decrease when descending from the root down to the leaves of $\mathcal{T}$. In the first level, p_0 distributes two matrices of size $n/2 \times n/2$ to the other process that is in charge to carry out A^TB tasks, and one sub-matrix of the same size to each of its four siblings that have to compute A^TA. For each level $l \in \{2, \ldots, \ell(P)\}$, the root process sends matrices of size $n/2^l$ to at most 7 other processes. Hence, during the distribution phase, $BW(n, P)$

is $O(5\,(n/2)^2 + 7 \cdot \sum_{l=2}^{\ell(P)} (n/2^l)^2) = O(5(n/2)^2 + 7/12\,n^2(1 - 1/4^{\ell(P)-2}))$. With similar considerations and taking into account the fact that processes sending symmetric portions of $\mathbf{C}$ only store its lower triangular part (low($\mathbf{C}$)), it holds that the bandwidth during the result retrieval phase amounts to $O((n/2)^2 + 4(n(n+2)/8) + 7 \cdot \sum_{l=2}^{\ell(P)} (n/2^l)^2) = O((n/2)^2 + n(n+2)/2 + 7/12\,n^2(1 - 1/4^{\ell(P)-2}))$. The thesis follows by summing together the two components. □

(a) Elapsed time.   (b) Effective GFLOPs.

Figure 3: AᴛA vs Intel MKL dsyrk

(a) Elapsed time.   (b) Effective GFLOPs.

Figure 4: Fᴀsᴛsᴛʀᴀssᴇɴ vs Intel MKL dgemm

From this analysis, we see that computation has the prominent role in time complexity $T(n, P) = C(n, P) + L(n, P) + BW(n, P)$. This fact will be confirmed by our experimental results, presented in Section 5, where we see how increasing the matrix sizes provides an always increasing benefit in using the distributed algorithm, proving that communication cost $L(n, P) + BW(n, P)$ is absorbed by the computational cost, $C(n, P)$, for growing values of n.

5 PERFORMANCE EVALUATION

We evaluate the performance of our algorithms with an extensive set of experiments over multiple benchmarks. Our code is available at https://github.com/filthynobleman/AtA.

5.1 Experimental Setup

All tests reported in this section were run on TeraStat[1], a cluster of 12 compute nodes, each equipped with 2 sockets of Intel Xeon E5-2630v3 8 cores, 2.4 Ghz, 4 GB RAM per core.

We test our algorithms and benchmark solutions on square and tall matrices, generated randomly. We carry out experiments in

[1]https://www.dss.uniroma1.it/en/node/6554

both single and double floating-point precision, to highlight the fact that our algorithm achieves good performance in both settings.

In the tests, we exploit the Intel Math Kernel Library (MKL) both by integrating BLAS routines for basic matrix operations, and for the validation of the proposed algorithms through performance comparisons with shared and distributed memory parallel benchmark solutions. MKL is a framework that includes routines and functions optimized for Intel and compatible processor-based computers, and provides C/C++ interfaces and the acceleration of libraries for Linear Algebra (including BLAS and ScaLapack) within several third-party math libraries. [16, 35].

5.2 Metrics

To compare the performance of our algorithms against benchmark methods, we use the average elapsed time in seconds and the effective GFLOPs. Effective GFLOPs is a measure for comparing classical and fast matrix-multiplication algorithms. For classical algorithms, which perform $2n^3$ floating point operations, Equation 9 gives the actual GFLOPs; for fast matrix-multiplication algorithms, it gives the performance relative to classical algorithms, but does not accurately represent the number of floating point operations performed [11]. For fair comparisons, we calculate the metrics as:

$$\text{effective GFLOPs} = \frac{rn^3}{\text{execution time in seconds} \cdot 10^9} \quad (9)$$

where $r = 1$ when we test algorithms specifically built for the $\mathbf{A}^T\mathbf{A}$ product, whereas $r = 2$ when algorithms for the general matrix multiplication are tested.

5.3 Sequential

Figures 3 and 4 show the execution time and effective GFLOPs of the sequential AᴛA and Fᴀsᴛsᴛʀᴀssᴇɴ routines, respectively. Their performance is compared to the Intel MKL counterparts: dsyrk and dgemm. The experiments are carried out on matrices of growing matrix size (from $2.5 \cdot 10^3$ to $2.5 \cdot 10^4$), and run on a single Intel core. The time difference between our solutions and the ones implemented by Intel MKL grows with the matrix size, reflecting the lower computational cost of our approach. Figure 4 proves how Strassen's algorithm benefits from the pre-memory-allocation strategy described in Section 3.3.

5.4 Shared memory

For evaluating the shared memory parallel implementation of the $\mathbf{A}^T\mathbf{A}$ product, AᴛA-S, we compare it against the Intel MKL implementation of the BLAS routine ssyrk, for single precision symmetric rank-K update. For both methods, we always use a 16 thread setup, and we analyse the execution time and the effective GFLOPs (Equation 9 with $r = 1$) while varying the number of available cores. In light of the sequential experiments shown in Figures 3 and 4, we compare AᴛA-S and MKL ssyrk on larger matrices, where tests highlight more interesting results. In particular, we run experiments on square matrices of size $3 \cdot 10^4 \times 3 \cdot 10^4$, $4 \cdot 10^4 \times 4 \cdot 10^4$ and on tall matrices of size $6 \cdot 10^4 \times 5 \cdot 10^3$. Figure 5 summarizes our results. As anticipated by the study of the computational complexity, the execution time is reduced by $1/4$ at each complete parallel level. Figures 5(a), 5(c) and 5(e) show how our algorithm can compete

(a) Elapsed time, $\mathbf{A} \in \mathbb{R}^{30K \times 30K}$.

(b) EGs, $\mathbf{A} \in \mathbb{R}^{30K \times 30K}$.

(c) Elapsed time, $\mathbf{A} \in \mathbb{R}^{40K \times 40K}$.

(d) EGs, $\mathbf{A} \in \mathbb{R}^{40K \times 40K}$.

(e) Elapsed time, $\mathbf{A} \in \mathbb{R}^{60K \times 5K}$.

(f) EGs, $\mathbf{A} \in \mathbb{R}^{60K \times 5K}$.

Figure 5: Experimental results of AtA-S and Intel MKL dsyrk in terms of elapsed time in seconds (left column) and effective GFLOPs (right column), varying the number of available cores P on fixed matrix sizes with a 16 threads configuration.

with the MKL implementation when the core availability is large, and that significantly outperforms the Intel implementation in the $P \leq 10$ cores setup. Furthermore, we show in Figures 5(b), 5(d) and 5(f) that AtA-S is capable not only of accomplishing a large amount of floating point operations per second, but also that its performance growth rate is consistent with the step-wise behaviour of the time complexity studied in Section 4.2.2. This justifies sporadic thinnings in performance gap between the two methods. From Figure 5, we can observe that the performance of both methods stall when more than 8 cores are used. Indeed, multi-threaded MKL automatically chooses the optimal number of threads (in our architecture, this corresponds to 16 threads). For a fair comparison, we use the same setup in AtA-S. Performance scales with the number of available cores, but, when hyper-threading is enabled, 8 cores are enough to reach the 16-thread plateau. Therefore, performance cannot increase significantly for $P > 8$.

5.5 Distributed memory

To complete our performance evaluation, we also compare our implementation for distributed architectures of AtA, AtA-D, with fast distributed algorithms for matrix multiplication. We recall

that AtA-D differs from standard methods for distributed matrix multiplication, as it does not perform computations on distributed matrices. Instead, in AtA-D the input matrix $\mathbf{A}$ is only stored by the root process, p_0, that first distributes it among other processes cooperating to perform the $\mathbf{A}^T\mathbf{A}$ product, and then collects the partial result of each process to combine them. This approach makes our method unsuitable for distributed chains of operations, since for every operation, the matrix must be repeatedly scattered and gathered back, thus introducing communication overhead, but our results highlight that it is an efficient alternative for distributing single $\mathbf{A}^T\mathbf{A}$ operations. At the current state-of-the-art, there are a variety of methods for multiplying distributed matrices, but in the most recent literature there are three algorithms which stand out:

(1) Intel MKL ScaLapack p?syrk: the Intel Math Kernel libraries (MKL) provide optimized implementation of ScaLapack routines for high-performance dense Linear Algebra operations on distributed clusters. In ScaLapack, distributed processes are organized in 2D grids of size $m_P \times n_P = P$. For each value of P, we set optimal m_P and n_P by calling `MPI_Dims_create`. We analyse the execution time required to perform the $\mathbf{A}^T\mathbf{A}$ matrix multiplication by the built-in ScaLapack function `pdsyrk`, and the time to retrieve the result of the operation.

(2) CAPS[2]: the Communication-Optimal Parallel Algorithm for Strassen's Matrix Multiplication [1] is a distributed algorithm for general square matrix multiplications $\mathbf{AB}$. Soon after CAPS, the same authors proposed CARMA [11], that also handles rectangular matrices. Nevertheless, it was not possible to test this method as it relies on Cilk Plus, a tool for parallel computing now marked as deprecated[3].

(3) COSMA[4]: differently from CAPS, this communication-optimal algorithm for general matrix multiplication does not rely on Strassen's algorithm, instead, it uses red-blue pebble game to precisely model the matrix-multiplication dependencies. In [26], the authors show that COSMA outperforms all previously proposed frameworks for general matrix multiplication. It also works for multiplication on transposed matrices, and therefore we test it to perform $\mathbf{A}^T\mathbf{B}$ products.

To simulate massively distributed architectures, in our experiments, we reserve only one core per distributed process. As a consequence, each process has small memory availability (4GB RAM/core). The results of our experiments for the distributed-memory solution are shown in Figure 6. In Figures 6(a), 6(d) and 6(g), marked lines represent the compute time of all considered methods. The shaded areas above the curves describing AtA-D and `pdsyrk` represent the additional time required for communication, i.e., for retrieving the resulting matrix to the root process. We consider two groups of square matrices, having size 10^4 and $2 \cdot 10^4$ (Figures 6(a), 6(b), 6(c) and 6(d), 6(e), 6(f) respectively), and one set of tall matrices of size $6 \cdot 10^4 \times 5 \cdot 10^3$ (Figures 6(g), 6(h), 6(i)). Because CAPS does not operate on rectangular matrices, we could not test it on the latter set of experimental configurations. As we can observe from Figure 6, scalability of AtA-D is nonlinear and it rather follows an almost-stepwise trend with respect to P. This is a consequence

---

[2]https://github.com/lipshitz/CAPS/

[3]https://www.cilkplus.org/, Last accessed 07-01-2021

[4]https://github.com/eth-cscs/COSMA

Figure 6: Experimental results of AtA-D, Intel MKL pdsyrk, CAPS and COSMA in terms of elapsed time in seconds (left column), effective GFLOPs (central column) and % of theoretical peak (right column) varying the number of distributed processes P on fixed matrix sizes.

of Equation 5, that shows how some values of P allow for a more effective and balanced workload between processes. This is evident for small values of P (when a greater availability of processes weighs significantly on the workload of each process), as well as for $P = 64$. Despite the different nature of the parallelism implemented in AtA-D with respect to the benchmark methods analysed in this section, our experiments corroborate the efficiency of the task distribution implemented in AtA-D. In Figures 6(c), 6(f) and 6(i) we show the percentage of theoretical peak performance (TPP) for all tested algorithms. We compute it as the effective GFLOPs over the theoretical performance peak of the nodes of our cluster. For all tested methods, the effective GFLOPs are computed as in Equation 9, (as those reported in Figures 6(b), 6(e), 6(h)), except for AtA-D, for which we now use the complexity of AtA (Equation 3). Regarding the percentage of theoretical peak, we can see how our algorithm has comparable behaviour with respect to the other solutions on square matrices, but it performs worse on the rectangular case. The high performance of our method relies on careful ordering and

placement of highly optimized BLAS routines. However, especially when working on tall matrices, we need to perform several calls to BLAS Level 1 routines (i.e., to compute intermediate sums both in AtA and FastStrassen) and system calls (i.e., memory copies) on very short rows. This leads to more memory accesses and poorer vectorization capability than dealing with the same amount of data, distributed in fewer, longer rows, would entail. As a consequence, the overall performance with respect to the theoretical peak is worsened. Furthermore, in Figures 6(c), 6(f) and 6(i) we see that AtA-D loses a bit of efficiency on larger matrices. This is due to the fact that each process calling the FastStrassen routine needs to allocate $3/2n^2$ space of memory, hence memory handling slows down the entire process. In addition, also in the last column of Figure 6, we can observe performance peaks after slow degradations as we did in the first two. We stress that this is a consequence of the fact that for some values of P, workload among processes is distributed more efficiently (see Equation 5). As a matter of fact, the computational complexity of AtA-D decreases exponentially at each level, but

n	SM (16 cores)	DM (96 cores)	Speed-up
30K	45.16 s	21.24 s	2.13
40K	106.19 s	43.96 s	2.42
50K	221.63 s	81.77 s	2.71
60K	863.32 s	129.08 s	6.69

Table 1: Shared memory (SM) vs distributed memory (DM) $A^T A$ implementation on large square $n \times n$ matrices.

the number of levels increases logarithmically with the number of processes, P. Therefore for numbers of processes that result in the same number of parallel levels, the improvement is less appreciable.

Finally, in order to study the scalability of AtA-D with respect to AtA-S, and to validate the possibility of integrating the two methods, we compare AtA-S and AtA-D on very large matrices of increasing size and report results in Table 1. AtA-S works on 16 cores with 16 threads, whereas AtA-D works on 6 distributed nodes, each equipped with 16 cores, for a total of 96 cores. Each node executes a distributed process calling either AtA-S for $A^T A$-type products, or multi-threaded MKL dgemm for $A^T B$-type multiplications. The times reported in Table 1 for AtA-D also include communication time (for distributing data and collecting results). Speed-up is computed as T_{SM}/T_{DM}, where T_{SM} and T_{DM} are the execution times of the shared and the distributed-memory algorithms, respectively. In accordance with our computational and communication cost analysis (Section 4.3.2), the speed-up of AtA-D over AtA-S increases when the size of the input matrix increases, as the computation cost overwhelms the communication overhead. Furthermore, the shared-memory implementation suffers when considering larger matrices, since frequent memory access slows down execution (two $60K \times 60K$ matrices require 57 GB out of the 64 GB available on the test machine), consequently decreasing performance. This is highlighted by the results of Table 1, where we can observe that $60K \times 60K$ matrices require high computation time, dominated by the time for memory management.

6 CONCLUSIONS

We propose AtA, an algorithm for the $A^T A$ product, that reduces the computational complexity of commonly employed algorithms, and that is conveniently implementable in practice on matrices defined on arbitrary domains and of any size and aspect ratio. The computational cost of AtA benefits from the fast matrix multiplication introduced by Strassen's algorithm, and is cache-oblivious. We show that AtA can be efficiently implemented in shared and distributed memory environments. In the shared memory implementation of AtA, tasks are assigned to parallel threads so that they work in perfect parallelism. Our theoretical analysis is supported by experiments that prove the excellent performance of our implementations in comparison with state-of-the-art counterparts.

ACKNOWLEDGMENTS

REFERENCES

[1] G. Ballard, J. Demmel, O. Holtz, B. Lipshitz, and O. Schwartz. 2012. Communication-optimal parallel algorithm for strassen's matrix multiplication. In *Proc. 24th ACM Symp. Parallelism in Algorithms and Architectures (SPAA)*. 193–204.

[2] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. 2011. Graph expansion and communication costs of fast matrix multiplication. In *Proc. 23rd ACM Symp. Parallelism in Algorithms and Architectures (SPAA)*. 1–12.

[3] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. 2013. Graph expansion and communication costs of fast matrix multiplication. *Journal of the ACM (JACM)* 59, 6 (2013), 1–23.

[4] A.R. Benson and G. Ballard. 2015. A Framework for Practical Parallel Fast Matrix Multiplication. In *Proc. 20th ACM SIGPLAN PPoPP*. 42–53.

[5] R. P. Brent. 1970. Algorithms for matrix multiplication. Tech. Rep. TR-CS-70-157, Stanford University.

[6] R. P. Brent. 1970. Error analysis of algorithms for matrix multiplication and triangular decomposition using Winograd's identity. *Numer. Math.* 16 (1970), 145–156.

[7] A. Charara, D. Keyes, and H. Ltaief. 2019. Batched triangular dense linear algebra kernels for very small matrix sizes on GPUs. *ACM Trans Math. Softw. (TOMS)* 45, 2 (2019), 1–28.

[8] A. Charara, H. Ltaief, and D. Keyes. 2016. Redesigning triangular dense matrix computations on GPUs. In *Euro-Par*. Springer, 477–489.

[9] D. Coppersmith and S. Winograd. 1987. Matrix multiplication via arithmetic progressions. In *Proc. 19th ACM Symp. Theory of Computing (STOC)*. 1–6.

[10] P. D'Alberto and A. Nicolau. 2007. Adaptive Strassen's matrix multiplication. In *Proc. 21st Int. Conf. on Supercomputing*. ACM, 284–292.

[11] J. Demmel, D. Eliahu, A. Fox, S. Kamil, B. Lipshitz, O. Schwartz O., and Spillinger. 2013. Communication-optimal parallel recursive rectangular matrix multiplication. In *IEEE 27th Int. Symp. on Parallel and Distributed Processing*. IEEE, 261–272.

[12] F. Desprez and F. Suter. 2004. Impact of mixed-parallelism on parallel implementations of the Strassen and Winograd matrix multiplication algorithms. *Concurr. Comput.: Pract. Exper.* 16, 8 (2004), 771–797.

[13] J. Dumas, C. Pernet, and A. Sedoglavic. 2020. On Fast Multiplication of a Matrix by Its Transpose. In *Proc. 45th Int. Symp. Symbolic and Algebraic Computation (Kalamata, Greece) (ISSAC '20)*. Association for Computing Machinery, New York, NY, USA, 162–169. https://doi.org/10.1145/3373207.3404021

[14] D. Eliahu, O. Spillinger, A. Fox, and J. Demmel. 2015. *Frpa: A framework for recursive parallel algorithms*. Technical Report UCB/EECS-2015-28. EECS Department, University of California, Berkeley.

[15] E. Elmroth, F. Gustavson, I. Jonsson, and B. Kågström. 2004. Recursive blocked algorithms and hybrid data structures for dense matrix library software. *SIAM review* 46, 1 (2004), 3–45.

[16] Developer Reference for Intel® Math Kernel Library C. 2019. (2019). https://software.intel.com/en-us/download/developer-reference-for-intel-math-kernel-library-c

[17] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. 1999. Cache-oblivious algorithms. In *40th Symp. Foundations of Computer Science (FOCS)*. IEEE, 285–297.

[18] F. Le Gall. 2014. Powers of tensors and fast matrix multiplication. In *Proc. 39th Int. Symp. Symbolic and Algebraic Computation*. 296–303.

[19] B. Grayson, A. Shah, and R. van de Geijn. 1995. A high performance parallel Strassen implementation. *Parallel Processing Letters* 6 (1995), 3–12.

[20] N.J. Higham. 1990. Exploiting fast matrix multiplication within the level 3 BLAS. *ACM Trans. Math. Softw.* 16, 4 (1990), 352–368.

[21] S. Hunold, T. Rauber, and G. Runger. 2008. Combining building blocks for parallel multi-level matrix multiplication. *Parallel Comput.* 34 (2008), 411–426.

[22] S. Huss-Lederman, E.M. Jacobson, A. Tsao, T. Turnbull, and J.R. Johnson. 1996. Implementation of Strassen's algorithm for matrix multiplication. In *Proc. ACM/IEEE Conf. on Supercomputing*.

[23] H. Jia-Wei and H. T. Kung. 1981. I/O Complexity: The Red-Blue Pebble Game. In *Proc. ACM Symp. Theory of Computing (STOC)* (Milwaukee, Wisconsin, USA). ACM, 326–333.

[24] M. Kadhum, M. H. Qasem, A. Sleit, and A. Sharieh. 2017. Efficient MapReduce matrix multiplication with optimized mapper set. In *Computer Science On-line Conference*. Springer, 186–196.

[25] Bo Kågström. 2004. Management of deep memory hierarchies–recursive blocked algorithms and hybrid data structures for dense matrix computations. In *Int. Workshop on Applied Parallel Computing*. Springer, 21–32.

[26] G. Kwasniewski, M. Kabić, M. Besta, J. VandeVondele, R. Solcà, and T. Hoefler. 2019. Red-Blue Pebbling Revisited: Near Optimal Parallel Matrix-Matrix Multiplication. In *Proc. Int. Conf. High Performance Computing, Networking, Storage and Analysis (SC '19)*. Article 24, 22 pages. https://doi.org/10.1145/3295500.3356181

[27] Q. Luo and J. Drake. 1995. A scalable parallel Strassen's matrix multiplication algorithm for distributed-memory computers. In *Proc. ACM Symp. Applied Computing, SAC'95*. 221–226.

[28] E. Peise and P. Bientinesi. 2017. Algorithm 979: recursive algorithms for dense linear algebra—the ReLAPACK collection. *ACM Trans. Math. Softw. (TOMS)* 44, 2 (2017), 1–19.

[29] M. H. Qasem, A. A. Sarhan, R. Qaddoura, and B. A. Mahafzah. 2017. Matrix multiplication of big data using mapreduce: a review. In *2nd Int. Conf. Applications of Information Technology in Developing Renewable Energy Processes & Systems*

(IT-DREPS). IEEE, 1–6.

[30] F. Song, J. Dongarra, and S. Moore. 2006. Experiments with Strassen's algorithm: From sequential to parallel. In *Proc. Parallel and Distributed Computing and Systems, (PDCS)*.

[31] A.J. Stothers. 2003. On the complexity of matrix multiplication. *Journal of Complexity* 19 (2003), 43–60. Issue 1.

[32] G. Strang. 2006. *Linear Algebra and Its Applications, Fourth Ed.* Thomson Brooks/Cole.

[33] V. Strassen. 1969. Gaussian elimination is not optimal. *Numerische mathematik* 13, 4 (1969), 354–356.

[34] M. Thottethodi, S. Chatterjee, and A.R. Lebeck. 1998. Tuning Strassen's matrix multiplication for memory efficiency. In *Proc. 1998 ACM/IEEE Conf. on Supercomputing (SC'98)*. IEEE, 36–36.

[35] E. Wang, Q. Zhang, B. Shenand G. Zhang, X. Lu, Q. Wu, and Y. Wang. 2014. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon Phi™*. Springer, 167–188.

[36] V.V. Williams. 2012. Multiplying matrices faster than Coppersmith-Winograd. In *Proc. 44th ACM Symp. Theory of Computing (STOC)*. 887–898.

[37] C-Q. Yang and B. P. Miller. 1988. Critical path analysis for the execution of parallel and distributed programs. In *Proc. 8th Int. Conf. on Distributed Computing Systems (ICDCS)*. IEEE, 366–373.

[38] W. Zeng, R. Guo, F. Luo, and X. Gu. 2012. Discrete heat kernel determines discrete Riemannian metric. *Graphical Models* 74, 4 (2012), 121–129.